ELSEVIER

Parallel Computing 27 (2001) 1417–1429

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Components and interfaces of a process management system for parallel programs ☆

Ralph Butler [a], William Gropp [b], Ewing Lusk [b,*]

[a] *Department of Computer Science, Middle Tennessee State University, 1301 E. Main Street, Murfreesboro, TN 37132, USA*

[b] *Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA*

Received 5 December 2000; received in revised form 2 April 2001

Abstract

Parallel jobs are different from sequential jobs and require a different type of process management. We present here a process management system for parallel programs such as those written using MPI. A primary goal of the system, which we call MPD (for multipurpose daemon), is to be scalable. By this we mean that startup of interactive parallel jobs comprising thousands of processes is quick, that signals can be quickly delivered to processes, and that `stdin`, `stdout`, and `stderr` are managed intuitively. Our primary target is parallel machines made up of clusters of SMPs, but the system is also useful in more tightly integrated environments. We describe how MPD enables fast startup and convenient runtime management of parallel jobs. We show how close control of `stdio` can support the easy implementation of a number of convenient system utilities, even a parallel debugger. We describe a simple but general interface that can be used to separate any process manager from a parallel library, which we use to keep MPD separate from MPICH. © 2001 Published by Elsevier Science B.V.

Keywords: Parallel job management; Process management

☆ This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, US Department of Energy, under Contract W-31-109-Eng-38.

* Corresponding author. Tel.: +708-252-7852; fax: +708-252-5986.

E-mail addresses: rbutler@mtsu.edu (R. Butler), gropp@mcs.anl.gov (W. Gropp), lusk@mcs.anl.gov (E. Lusk).

1. Introduction

A parallel programming environment may be viewed as comprising three interacting components: a *job scheduler*, which decides what resources a parallel job consisting of multiple processes will run on; a *process manager*, which starts and terminates processes and provides them with a number of services; and a *parallel library* such as MPI, which a parallel application calls upon for communications. Since these components need to communicate with one another, they are often integrated into a single system. An important research question is to what extent they can be separated from one another with well-defined interfaces so that they can be independently developed. A further research question is whether the resulting system can be made scalable to jobs involving thousands of communicating processes. In this paper we focus on the process manager component. We describe a design and an implementation we call MPD (for multipurpose daemon) that provides both fast startup of parallel jobs and a flexible run-time environment that supports parallel libraries through a small, general interface.

A parallel job is both similar to a sequential job and different from one in significant ways. Resource allocation and scheduling are considerably more complex, and we do not address these issues here. But for process management issues alone, complexity arises from the fact that there may be multiple executables, multiple sets of command-line arguments, even different environments for different processes. Task farm jobs are different from true parallel jobs in which processes will communicate with one another, not just with a master process. On clusters we must either set up all connections ahead of time or provide some way for a process needing to establish communication with another process to ask for help from the process manager (who is the only one who knows where that other process is). The first approach is not scalable to large number of processes, and in scalable applications not all such connections will eventually be needed; therefore, a process manager must be able to provide information services to parallel jobs to allow connections to be made dynamically. Scalable startup is needed to make interactive parallel jobs feasible. Parallel jobs also need scalable signal delivery and a reasonable semantics for `stdio` redirection.

We assume familiarity with process management for sequential jobs on Unix. Components of process management we take to be the process id, executable name, environment variables, command-line arguments, signals (especially `cntl-C`, `cntl-Z`, and resume signals), `stdin`, `stdout`, and `stderr` and their redirection. We differentiate process management from *scheduling*, which is the problem of locating resources and a time to use them. Batch schedulers often combine scheduling with process management.

In Section 2 we summarize related work. In Section 3 we state our explicit design goals, how these goals lead to implementation decisions, and interesting features of the resulting system, including how it can be used to create a parallel debugger out of an existing single-process debugger. Section 4 briefly describes a general-purpose interface between process managers and parallel libraries and how this interface is implemented on the process manager side by MPD. Section 5 summarizes preliminary experiments that make us optimistic about the usefulness of MPD as a process manager for large-scale systems. We conclude with a summary of progress to date and a description of our future plans.

The MPD system is in use and is available as open source as part of the MPICH system [19], obtainable from http://www.mcs.anl.gov/mpi/mpich.

An abbreviated report on this work appeared in [9]. Here we provide more details than was possible there, describe new additions to the system, and outline an interface that can be used by parallel programs to obtain services from a process manager like MPD.

2. Related work

All parallel computing environments that support execution of truly parallel programs (those in which any two processes can communicate with one another) have had to address at least some of the issues that we address with MPD. Parallel programming systems, such as PVM [15, p. 4], [10], and implementations of MPI such as MPICH [19] and LAM [8] all provide some mechanism for starting and running parallel programs, often with a specialized daemon process. MPD differs from these systems in two ways. First, it is independent of any particular programming library, instead implementing a simple interface (described in Section 4) by which any library, including these, can make use of its services. Second, it is designed specifically to enable rapid startup of jobs consisting of hundreds to thousands of processes.

Many systems are intended to manage a collection of computing resources for both single-process and parallel jobs; see the survey by Baker et al. [4]. Typically, these use a daemon that manages individual processes, with emphasis on jobs involving only a single process. Widely used systems include PBS [23], LSF [24], DQS [12], and Loadleveler/POE [20]. The Condor system [21] is also widely used and supports parallel programs that use PVM [25] or MPI [17,26]. The PCT daemons [7] used on the Sandia Cplant machine implement a tree-based scalable startup, in addition to other services. More specialized systems, such as MOSIX [5] and GLUnix [16], provide single-system image support for clusters. Harness [6,22] shares with MPD the goal of supporting management of parallel jobs. Its primary research goal is to demonstrate the flexibility of the "plug-in" approach to application design, potentially providing a wide range of services. The MPD system focuses more specifically on the design and implementation of services required for process management of parallel jobs, including high-speed startup of large parallel jobs on clusters and scalable standard I/O management. The book [14] provides a good overview of metacomputing systems and issues, and Feitelson [13] surveys support for scheduling parallel processes.

3. Design of MPD

In this section we describe our goals in constructing MPD and outline the system's architecture.

3.1. Goals

Several explicit goals have governed the design of the MPD system.

Simplicity: The persistent (across jobs) part of the system should be simple and robust. This part of the system should be runnable as root. If its behavior is not completely transparent, we will never be able to convince system administrators to run the daemons as root.

Speed: Startup of parallel jobs should be quick enough to provide an interactive "feel", so that large but short jobs make sense. Large (in the number of processes) but short (in time) characterizes system utilities such as those described in [18]. Our immediate target is to start 1000 processes in a few seconds, while still providing a way for such processes to establish contact with one another. Our long-term goal is to support management of 10,000 processes.

Robustness: The persistent part of the system should be at least moderately fault tolerant. An unexpected crash of one machine should not bring down the whole system. There should be no single "master" process.

Scalability: The complexity or size of any component should not depend on the number of components.

Individual process environments: It should be possible to start a parallel job in which the executable files, environment variables, and command-line arguments are different for each process. It should be possible to collect return codes individually from processes.

Collective identity of a parallel job: It should be possible to treat a parallel job as a single entity that can be suspended, continued (signaled, in general), or killed collectively as if it were a single process. The system should manage `stdin`, `stdout`, and `stderr` in a useful and scalable way and allow them to be redirected as if the parallel job were a single process. An important component of a job's collective identity is its *termination*. All resources allocated for the job, such as files, System V IPC's, other processes, etc., must be reliably freed, even if the job terminates abnormally.

It is explicitly not a goal of the MPD system to provide scheduling services, which we believe to be a separate function from process management. We expect the decision on precisely which resources to use to run a job to be made by a separate scheduler, which will then communicate its decision to the process management system. Design of the interface by which this occurs is an interesting problem, but not addressed here. Note that many existing systems combine scheduling and process management, an organization that we find limits flexibility. In this paper we focus solely on process management.

3.2. Deriving the design from the goals

The goals of simplicity and robustness lead us to adopt a multicomponent system. The *daemon* itself is persistent (may run for weeks or months at a time, starting many jobs), typically one instance per host in a TCP-connected network. *Manager* processes will be started by the daemons to control the application processes (*clients*) of a single parallel job and will provide most of the MPD features. The goal of speed requires that the daemons be in contact with one another prior to job startup, and the goals of scalability and "no master" suggest that the daemons be connected in a ring. [1] The services that the managers will provide (see Section 3.3) suggest that they be in contact as well, and the fastest way for them to form these connections is to inherit part of the ring connectivity of the daemons. Separate managers for each user process support the individual process environments. The goal of having a collective identity for a parallel job leads us to treat the `mpirun` process as such a representative and use it to deliver signals and `stdin` to application processes and collect `stdout` and `stderr` output from them. This suggests that the `mpirun` process connects first to the daemon ring in order to start the job, and then switches the connection to the manager ring in order to control the job. The goal of speed suggests that these latter connections be restricted to a process running on the same host, either the daemon itself or a persistent gateway process if the daemon is run as root, so that authentication can be through the file system (a Unix rather than a network socket). We refer to all such processes as *console commands*. The console commands `mpd`, `mpdtrace`, and `mpdallexit` manage the daemons themselves; `mpdmpexec` and `mpirun` start parallel jobs; and `mpdlistjobs`, `mpdkilljob`, and `mpd-gangjobs` help to manage parallel jobs. There are a few others, and it is easy to write new console commands as needed. Finally, in order that this infrastructure be available to support MPI programs or other parallel tools, there needs to be a *client library* that each application process may use to interact with its manager.

We do not specify how the daemons are started or connected, since the system provides a number of alternatives, and the process need not be particularly fast. A console command is started by the user, either interactively or under the control of a batch scheduler. The daemons `fork` and `exec` the managers, which use information given them by the daemons to connect themselves into a ring, then `fork` and `exec` the clients. The startup messages traverse the ring quickly, so most `fork`ing, `ex-ecing`, and connecting takes place in parallel, leading to fast startup even for large jobs. The situation is then as shown in Fig. 1, where the clients may be application MPI processes. Solid lines represent sockets, except for the vertical ones, which represent pipes. The dashed lines represent the trees of connections for forwarding `stdout` and `stderr`, and the dotted lines represent *potential* connections among the client processes. The dot-dashed line is the original connection from console to local daemon on a Unix socket, which is replaced during startup by the network connection to the first manager.

3.3. Interesting features

Space restrictions prevent a complete description of all the features and capabilities of the MPD system, but in this section we mention a few highlights.

[1] While a ring is not ultimately scalable, it is more so than the typical star used in many process management systems, and our experiments have shown it feasible for the thousand-daemon domain.

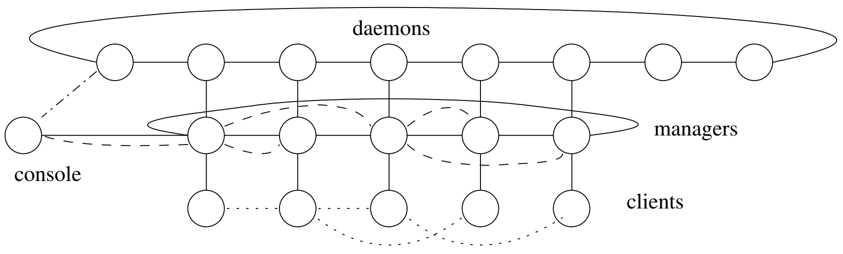

Fig. 1. Daemons with console process, managers, and clients.

Security: Whenever a process advertises a “listener” socket and accepts connections on it, the possibility exists that an unknown or even malicious process will connect. This is particularly dangerous if the process accepting the connection can start processes as the MPD daemon can. We currently use the “challenge-response” system described in [27]. In the long run, we expect to modify this component of the system to use more elaborate schemes and extend them to other connections such as client/gateway authentication. This will have little impact on the job startup speed, since the daemon component startup is separate from job startup.

Fault tolerance: If a daemon dies, this fact is detected and the ring is reknit. This provides a minimal sort of fault tolerance, since the ring remains intact. A new MPD daemon can be inserted in the ring where the old one was, but this process is not (yet) automatic. We use two methods for detecting the death of a daemon. Each daemon is responsible for detecting the death of its neighbor to the right in the ring in two ways. The quickest indication is receipt of EOF on the socket connecting it to its neighbor. A slower but more reliable method is that each process periodically sends a message to its right-hand neighbor, and assumes that the neighbor has died if an acknowledgment is not received in a timely fashion.

Signals: Signals can be delivered to client processes by their managers. We currently use this capability in three specific ways. First, signals delivered to a console process are propagated to the clients, so that a parallel application as a whole can be suspended with `cntl-Z`, continued, and killed with `cntl-C`, just as if it were a single process. Second, in the `ch_p4mpd` device in the MPICH implementation of MPI, client processes can interrupt one another with requests to dynamically establish client-to-client connections. Such requests go up into the manager ring from the originating client, around the ring to the manager of the target process, which signals its client. Third, a separate console process can be used to implement a simple but effective gang scheduler. Gang scheduling is the process of ensuring that all the processes of a parallel job are swapped in and scheduled to run at the same time across a collection of machines; it is particularly important for jobs that contain a large number of collective operations. MPD can provide gang scheduling by using signals to pause and resume parallel jobs running on the same collection of machines. This mechanism is also useful for pausing long-running parallel jobs to run short parallel utility programs.

Support for MPI implementations: MPD provides a version of the BNR library described in Section 4. Version 1.2.1 of MPICH makes calls to this library to find out a process's rank, where other processes are and how to contact them, and so forth.

On clusters of SMPs, it is easy to specify that multiple processes are to be started on the same machine and share memory. Specifically, `mpirun -np 180 -g 2 a. out` starts processes in groups of two and places in their environment a key that can be used to acquire group-attached shared memory and other information needed to set up multimethod communication for an MPI implementation. We use this technique on the Chiba City cluster [1] at Argonne, which has dual CPU nodes.

Handling standard I/O: Managers capture the `stdout` and `stderr` of their clients and forward them up a pair of binary trees of socket connections, each manager merging `stdout` and `stderr` from its client with that from each of its two children. A command line option tells the managers to provide a rank label on each line of output from their clients. A pedestrian but useful application of this feature is that it helps with programs that may not have been written to be parallel in the first place. Both standard output and error output are automatically identified with their source process without touching the original code. The feature is also useful when invoking system utilities in "task farm" mode. The command

```
mpdmpexec -np 128 ps auxww | grep mpdman
```

finds (in conjunction with the use of `hostname`) where `mpd`'s have started managers.

Standard input (to `mpirun`, for example) by default is delivered to the client managed by manager 0. This seems to be what most MPI users expect and what most MPI implementations do. (The MPI standard does not specify.) However, control messages can be used to change this behavior to direct `stdin` to any specific client or broadcast it to all clients.

Environment variables: By default the `DISPLAY` environment variable of the shell in which `mpirun` is invoked is forwarded to the managers and set for the clients. This allows clients to use *X* graphics. We plan to replace this non-scalable approach to graphical output with one similar to the one used for `stdout` and `stderr`. Other environment variables can be specified on the command line for propagating to the application processes.

Client wrapping: The semantics of the Unix `fork` and `exec` system calls provide useful benefits. When a manager `forks` a client process, for example, it first sets up the manager–client pipes for control messages and standard I/O. The "lower" ends of these pipes are inherited by any process that the client forks. Thus, even though the client is not using any of the client library, managers can manage clients that themselves run the "real" application process. We call this scheme *client wrapping*. Thus `mpirun -np 16 nice -5 myprog` lowers the priority of a parallel job to be run on one's colleagues' workstations, and `mpirun -np 16 pty myprog` can be used when `myprog` needs to be attached to a terminal (otherwise our capture of `stdin` and `stdout` modifies their buffering behavior). The program `pty` is distributed with the MPD system.

Putting it all together: The combination of I/O management, especially redirection of `stdin`, line labels on `stdout`, and client wrapping can be surprisingly powerful. We have used these features of the MPD system to add an option to `mpirun` that invokes `gdb` as a client wrapper and dynamically redirects `stdin`. While `mpirun -np 3 cpi` runs `cpi` directly as an MPI job, `mpirun -np 3 -d cpi` runs each `cpi` process under the control of (wrapped by) the `gdb` debugger. (Other sequential debuggers could be used, but are not yet supported.) Thus multiple instances of `gdb` are being run. Output of the `gdb`'s is labeled by process rank. The "`(gdb)`" prompts are intercepted by the `mpirun` process and counted, so that it can issue an "`(mpigdb)`" prompt when one has been received from each process. When possible, identical lines from multiple processes are merged, for scalability. In addition, `mpirun -d` uses the "`z`" command (one of the few single letters not already claimed by `gdb`) to redirect `stdin` to a specific `gdb` instance or to all processes. Thus processes can be stepped and breakpoints can be set either collectively or individually, and collectively printing a variable will provide all values with rank labels. The following is an example of how this works:

```
donner% mpirun -np 5 -d cpi      # default is stdin bcast
(mpigdb) b 29                    # set breakpoint for all
0-4: Breakpoint 1 at 0x8049e93: file cpi.c, line 29.
(mpigdb) r                       # run all
0-4: Starting program: /home/lusk/mpich/examples/basic/cpi
0: Breakpoint 1, main (argc=1, argv=0xbffffa84) at cpi.c:29
1-4: Breakpoint 1, main (argc=1, argv=0xbffffa74) at
cpi.c:29
0-4: 29  n=0;                    # all reach breakpoint
(mpigdb) n                       # single step all
0: 38    if (n==0) n=100; else n=0;
1-4: 42    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
(mpigdb) z 0                     # limit stdin to rank 0
(mpigdb) n                       # single step process 0
0: 40    startwtime=MPI_Wtime();
(mpigdb) n                       # until caught up
0: 42    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
(mpigdb) z                       # go back to bcast stdin
(mpigdb) n                       # single step all
              ...                # until interesting spot
(mpigdb) n
0-4: 52    x=h * ((double)i - 0.5);
(mpigdb) p x                     # bcast print command
0: $1=0.0050000000000000001      # 0's value of x
1: $1=0.014999999999999999       # 1's value of x
2: $1=0.025000000000000001       # 2's value of x
3: $1=0.035000000000000003       # 3's value of x
```

```
4: $1 = 0.044999999999999998      # 4's value of x
(mpigdb) c                        # continue all
0: pi is approximately 3.141600986923, Error is
0.000008333333
0-4: Program exited normally.
(mpigdb) q                        # quit
donner %
```

Running the daemons as root: By default, the MPD daemons are run in ordinary user mode. This is useful for development, but in production we do not wish the machines to fill up with `mpd` processes being run by various users; we prefer to have only one `mpd` per machine. To this end the daemons can be configured to be run as root. In this situation the console is a `setuid` program that runs as root only to connect briefly to the local `mpd`, then reassumes the user's user id, group id, and group memberships and sends these to the `mpd`, so that the managers and clients run as the user in every way. In this mode the daemon is running as a "true" daemon, detached from any specific terminal and logging information and error messages to `syslog`.

Interface for "real" parallel debuggers: MPD and the BNR interface described in Section 4 cooperate to implement the debugger interface described in [11]. Thus if the clients are linked with the MPICH library, or any MPI library that provides the library side of the interface, then any debugger (such as TotalView [3]) that uses this interface can be used to debug parallel jobs started by the MPD daemons. We again use the model that the `mpirun` process is a representative of the entire parallel job. Thus to run under the debugger one does `totalview mpirun -np 10 a.out`, and to attach to a running parallel job, one brings up TotalView and tells it to attach to the running `mpirun` process. Then TotalView starts its servers (optionally with help from MPD) and they attach to the client processes. Then the usual TotalView Debugging commands can be used, including examination of the message queues.

A particular advantage of being able to attach to running processes is that many timing and synchronization errors can hide from sight when a job is run under the debugger from the beginning. Especially when such an error causes a parallel job to hang, being able to attach to its processes and examine the state of each process and the message queues is invaluable.

4. A general process manager interface

One reason for creating MPD was that no existing process manager really had the needed support for parallel jobs that we could use for MPICH. One research goal of the project was to determine a minimal set of services that would need to be added to an existing commercial or open source process manager in order to provide what was needed by a parallel library to implement MPI, especially MPI-2, with its dynamic process creation and one-sided operations. In this section we describe such an interface and how it is used by MPICH and implemented by

MPD. The interface decouples MPD from MPICH, allowing MPICH to be used with any process manager that implements this simple interface and allowing other sorts of parallel systems besides MPICH to be supported by the MPD runtime system.

In Section 1 we mentioned that we consider the scheduler, process manager, and parallel programming library to be separate components of a parallel environment. We have tried to isolate and simplify the interface between the process manager and the parallel programming library into a simple specification called BNR.

BNR is the interface by which the parallel programming library (an example is MPICH) obtains information from the process manager (an example is MPD) that only the process manager initially knows, such as the rank in the parallel job of an individual process and the information necessary for a process to dynamically forge a connection with a process with a different rank.

The central component of BNR is a primitive database interface consisting of job-local `put`, `get`, and `fence` calls, by which processes can place keyword = value data into the database, retrieve it by keyword, and coordinate with the other processes in a parallel job to ensure that expected data are deposited before it is accessed.

We use this particular interface for scalability. The use of the `fence` primitive permits a single synchronization operation after which all data that have been `put` before the `fence` can be retrieved by a `get`. The `get`, `put`, and `fence` calls are local to *groups* of processes within jobs, which in an MPI implementation can correspond to MPI groups. Groups play an important role in libraries that use dynamic process creation calls (such as `MPI_Comm_Spawn`).

Version 1 of the BNR API, which is all that is necessary to support MPI-1 implementations, is shown here in its entirety, thus illustrating its simplicity. Version 2 will include the more complex group management routines necessary to support spawning of new processes and connection of existing groups, as well as the routines necessary for a full implementation of MPI-2.

```
int BNR_Init(BNR_gid *mygid);
int BNR_Put(BNR_gid gid, char *attr, char *val);
int BNR_Fence(BNR_gid gid);
int BNR_Get(BNR_gid gid, char *attr, char *val);
int BNR_Finalize();
int BNR_Rank(BNR_gid group, int *myrank);
int BNR_Nprocs(BNR_gid group, int *nprocs);
```

MPD implements the BNR interface by keeping the database distributed among the managers for a job and routing data access requests around the manager ring as necessary. In the `ch_p4mpd` device in the current version of MPICH, we use only this interface in the implementation, so that any process manager implementing this interface can be used to manage MPICH programs without knowing any MPICH internals.

Version 2 of BNR will be the interface by which the parallel program library requests more complex actions on the part of the process manager. A typical request would be to start processes, either as part of MPI-2's `MPI_Comm_spawn` or with `mpirun` or `mpiexec`.

5. Experiments

Most development of MPD has been on workstation networks where startup of 32-process jobs on five workstations is virtually instantaneous, compared with the approximately 1.5 s per process required by the `ch_p4` version of MPICH. An early test of the feasibility of using the ring topology showed that a message could make 1024 hops around the ring in less than 0.4 s, which gave us confidence that the ring would not impose scalability limits, at least in the near term. Recently we began experiments on Chiba City, a Linux testbed for parallel computer science research [1]. We performed one set of tests on 211 nodes connected by Fast Ethernet. Because we were interested only in process startup time, we tested execution of trivial parallel jobs, for example,

```
time mpirun -np 211 hostname
time mpirun -np 422 -g 2 hostname
```

We found that starting 211 processes (one on each node) and collecting the `stdout` output of `hostname` took about 2 s to execute. Starting twice as many processes (one for each CPU) took about 3.5 s, including setting up the relatively complex `stdout` tree and collecting the output. Sending a message around the ring of 211 MPD daemons took only 0.13 s.

MPD is now running on our Chiba City cluster in root mode, serving as an experimental production process manager. We have added facilities that allow the MPD daemons to start jobs linked with Myricom's MPI implementation, MPICH-GM, so that MPI jobs can be started with MPD and run over Chiba City's Myrinet network.

6. Future development

The existing MPD system, consisting of daemons, managers, console commands, and client library, meets our goals of simplicity, robustness, and scalability. It is used for fast startup of MPI jobs and others on systems with hundreds of machines. The flexibility of its `stdio` control mechanism has provided unexpected benefits, such as a "poor man's" parallel debugger. It meets our goals for the collective identity of a parallel job.

In the near term, we expect to use the system to implement the dynamic process creation part of MPI-2 in MPICH. We will do this by expanding the BNR interface to provide the necessary functions and implementing the expanded interface in MPD. The existing group-based put/get/fence interface will work well with the more dynamic group creation that will be needed to support spawning of new processes and connecting existing groups of processes together. We are working on a precise definition of how MPD will interoperate with a full-featured scheduling system such as the Maui scheduler [2]. We believe that the MPD daemons can also begin to provide more services, such as run-time performance monitoring.

In the long run, as machines grow from hundreds to thousands of nodes, our rings of daemons and managers may have to grow into a more sophisticated structure, such as rings of rings, in order to continue to provide fast startup. We anticipate that this can be done without substantially changing the MPD design presented here. We will also need a more sophisticated output merger in order to provide scalable `stdout`, for example for large-scale parallel debugging.

In summary, we are finding the MPD system already a useful contribution to one's parallel programming environment and expect its applicability to expand in the near future. We also view its design as a valuable starting point for future research into large-scale parallel job execution environments.

References

[1] Chiba City home page, http://www.mcs.anl.gov/chiba.

[2] The Maui scheduler home page, http://maui-scheduler.mhpcc.edu/new_doc, http://www.mhpcc.edu/maui.

[3] TotalView Multiprocess Debugger/Analyzer, 2000, http://www.etnus.com/Products/TotalView.

[4] M.A. Baker, G.C. Fox, H.W. Yau, Review of cluster management software, NHSE Rev. 1 (1) (1996).

[5] A. Barak, S. Guday, R.G. Wheeler, The MOSIX distributed operating system: load balancing for UNIX, vol. 672, in: Lecture Notes in Computer Science, Springer, New York, 1993.

[6] M. Beck, J.J. Dongarra, G.E. Fagg, G. AlGeist, P. Gray, J. Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulous, S.L. Scott, V. Sunderam, HARNESS: a next generation distributed virtual machine, Int. J. Future Generation Comput. Syst. 15 (5/6) (1999).

[7] R. Brightwell, L.A. Fisk, Scalable parallel application launch on Cplant, 2001, preprint.

[8] G. Burns, R. Daoud, J. Vaigl, LAM: an open cluster environment for MPI, in: J.W. Ross (Ed.), Proceedings of Supercomputing Symposium '94, University of Toronto, 1994, pp. 379–386.

[9] R. Butler, W. Gropp, E. Lusk, A scalable process-management environment for parallel programs, in: J. Dongarra, P. Kacsuk, N. Podhorszki (Eds.), Recent Advances in Parallel Virutal Machine and Message Passing Interface, no. 1908 in Springer Lecture Notes in Computer Science, September 2000, pp. 168–175.

[10] R. Butler, E. Lusk, Monitors, messages, and clusters: the p4 parallel programming system, Parallel Comput. 20 (1994) 547–564.

[11] J. Cownie, W. Gropp, A standard interface for debugger access to message queue information in MPI, in: J. Dongarra, E. Luque, T. Margalef (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, vol. 1697 of Lecture Notes in Computer Science, Springer Verlag, 1999, pp. 51–58, in: Proceedings of the Sixth European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 1999.

[12] DQS home page, http://www.scri.fsu.edu/~pasko/dqs.html.

[13] D.G. Feitelson, A Survey of Scheduling in Multiprogrammed Parallel Systems, Research report rc 19790 (87657), IBM T.J. Watson Research Center, Yorktown Heights, NY, February 1995.

[14] I. Foster, C. Kesselman (Eds.), The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, Los Altos, CA, 1999.

[15] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, V. Sunderam, PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Network Parallel Computing, MIT Press, Cambridge, MA, 1994.

[16] D.P. Ghormley, D. Petrou, S.H. Rodrigues, A.M. Vahdat, T.E. Anderson, GLUnix: a global layer Unix for a network of workstations, Software – Practice and Experience 28 (9) (1998) 929–961.

[17] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, M. Snir, MPI – The Complete Reference: Vol. 2, The MPI-2 Extensions, MIT Press, Cambridge, MA, 1998.

[18] W. Gropp, E. Lusk, Scalable Unix tools on parallel processors, in: Proceedings of the Scalable High-Performance Computing Conference, IEEE Computer Society Press, 1994, pp. 56–62.

[19] W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message–passing interface standard, Parallel Comput. 22 (6) (1996) 789–828.

[20] IBM, Loadleveler: using and administering, version 2 release 1 Ed., November 1998, SA22-7311-00.

[21] M.J. Litzkow, M. Livny, M.W. Mutka, Condor – a hunter of idle workstations, in: Proceedings of the Eighth International Conference on Distributed Computing Systems, San Jose, CA, June 1988, pp. 104–111.

[22] M. Migliardi, V. Sunderam, PVM emulation in the Harness metacomputing system: a plug-in based approach, in: J.J. Dongarra, E. Luque, T. Margalef (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface: Sixth European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 26–29, 1999: Proceedings, vol. 1697 of Lecture Notes in Computer Science, Springer, Berlin, 1999, pp. 117–124.

[23] PBS home page, http://pbs.mrj.com/.

[24] Load sharing facility (LSF), http://www.platform.com.

[25] J. Pruyne, M. Livny, Interfacing condor and PVM to harness the cycles of workstation clusters, Future Generation Computer Systems 12 (1) (1996) 67–85.

[26] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra, MPI – The Complete Reference: The MPI Core, second ed., vol. 1, MIT Press, Cambridge, MA, 1998.

[27] A.S. Tanenbaum, Computer Networks, third ed. Prentice-Hall, Englewood Cliffs, NY.